Representation and Presentation of Requirements Knowledge

W. Lewis Johnson, Martin S. Feather, and David R. Harris

***Abstract*—This paper describes the approach to representation and presentation of knowledge used in ARIES, an environment to experiment with support for analysts in modeling target domains and in entering and formalizing system requirements. To effectively do this, ARIES must manage a variety of notations so that analysts can enter information in a natural manner, and ARIES can present it back in different notations and from different viewpoints.**

To provide this functionality we use a *single*, highly expressive, internal representation for all information in the system. Our system architecture separates representation and presentation, in order to localize consistency and propagation issues. The presentation architecture is tailored to be flexible enough so that we can easily introduce new notations on top of the underlying representation. We have coupled presentation knowledge to specification evolution knowledge, thereby leveraging common representations for both in order to provide automated focusing support to users who need informative guidance in creating and modifying specifications.

***Index Terms*—Knowledge-based software engineering, requirements analysis, software presentation, software reuse, software specification, transformations.**

I. Introduction

We have built an experimental requirements/specification environment called ARIES[1] to investigate the support of requirements analysts in evaluating system requirements and codifying them in formal specifications. Our purpose has been to explore representation and reasoning issues in four key areas: presentation, reuse, reasoning, and evolution. Each of these is, we believe, essential for providing automated assistance to requirements engineering.

The status of ARIES is that of an experimental system—it serves as a testbed for study of the facilities that a real system would, we think, have to provide. As such, we have *not* demonstrated the utility of ARIES by having real-life analysts use it to support their actual activities. Rather, we ask the reader to share our assumption that a system something like ARIES, in particular, a system with capabilities for representation and reasoning akin to those that we have built into ARIES, would prove useful. The focus of this paper is on the means by which we provide such capabilities.

The essence of our approach is to use a single highly expressive internal representation for all information in the system, and have available a variety of "presentations" for viewing aspects of that information. In addition to simply viewing information, we can also enter information into the knowledge base via many of these presentations, and modify existing information by direct manipulation of presentations.

The rest of this paper is structured as follows.

Section II describes the broader context of our work, namely the goal of providing integrated knowledge-based support for all aspects of the software life cycle. Our part of this is to provide support for all aspects of requirements and specification analysis. This leads to the assumptions that underlie our design of ARIES, whose capabilities are summarized briefly. Section III shows the analyst's view of ARIES through a detailed example of analyst/ARIES interaction. Of particular interest are the ways in which the analyst is permitted access to the requirements knowledge through presentations tailored to the aspects that he/she would wish to focus on. Section IV describes the knowledge representation that we use, and the facilities that allow us to easily build upon this representation. This extensibility is crucial to the success of our approach. Section V describes how we build the presentations that the analyst sees. We have striven to make these presentations easy to construct and modify, so that we may readily experiment with them, and add new ones as needed. Section VI briefly outlines other aspects of our system, namely the facilities it provides to support the analyst in evolving and validating requirements. These also rely upon the capabilities embodied in our internal representation and its associated mechanisms. Section VII contrasts and compares our approach to related efforts, and Section VIII summarizes our approach and findings.

II. Assumptions and Background

Our design decisions for ARIES follow from a perspective on the requirements/specification process and on the modes of analyst–system interaction necessary for effective automated support. We begin by placing ARIES in the context of a broader effort toward supporting software development, then briefly outline our perspective (a more thorough discussion of which has been published elsewhere [26], [27]), and finally discuss

Manuscript received December 1, 1991; revised July 1, 1992. This work was sponsored in part by the Air Force Systems Command, Rome Air Development Center, under contracts F30602-85-C-0221 and F30602-89-C-0103, and in part by the Defense Advanced Research Projects Agency under Contract NCC-2-520. Views and conclusions contained in this paper are the authors' and should not be interpreted as representing the official opinion or policy of the U.S. Government or any agency thereof. Recommended by M. Jarke and A. Borgida.

L. Johnson and M. Feather are with USC/Information Sciences Institute, Marina del Rey, CA 90292-6695.

David Harris is with Lockheed Sanders, Signal Processing Center of Technology, Nashua, NH 03061-0868.

IEEE Log Number 9202816.

[1] ARIES stands for Acquisition of Requirements and Incremental Evolution of Specifications.

0162–8828/92$03.00 © 1992 IEEE

how and why we have constructed ARIES to support this requirements/specification process.

2.1. *The Broader Context of Software Development*

The ongoing Knowledge-Based Software Assistant (KBSA) program, of which ARIES is a product, was conceived as an integrated knowledge-based system to support all aspects of the software life cycle [17]. Such a system would offer specification-based software development in which efficient implementations are mechanically derived from executable specifications. ARIES's role within this effort is to support the requirements and specification aspects of the software lifecycle; one of the outputs of ARIES is a specification in the ERSLA language [43], ready for the follow-on stage of mechanical optimization. (ERSLA is Andersen Consulting's extension of the very-high-level programming language Refine [36], adding more specification constructs to Refine).

ARIES builds on the results of earlier efforts: requirements analysis, studied in Lockheed Sanders's Knowledge-Based Requirements Assistant [21], and specification construction, validation, and evolution, studied in ISI's Knowledge-Based Specification Assistant [23], [24], [35].

2.2 *Our Perspective on the Requirements/Specification Process*

We envision a requirements/specification process which results in software requirements specifications—descriptions at whatever level of detail or formality is necessary—of systems to be built. Our vision is of a seamless process in which there is no distinction between "requirements" and "specifications." Our goal is to support the analyst in his/her major activities of this process, which we identify as the following.

- *Acquisition:* the gathering of information about the problem. Analysts collect such information from a variety of sources, including interviews with clients, and source documents describing the domain in which the system will operate.
- *Reasoning:* the analysis and combination of gathered information. Analysts need to understand and reason about large specifications, propagate design decisions throughout the emerging system description, and merge possibly conflicting requirements that have been acquired from groups of analysts working simultaneously.
- *Evolution:* the modification of requirements and specifications. The analyst's changing understanding of requirements leads to the desire to evolve the description. This may be a response to altered circumstances, a correction of inaccurate or overly ideal requirements, or a desire to make use of similar pre-existing requirements.
- *Presentation:* the communication of gathered requirements. The fundamental goal of communication is to pass information to designers and stakeholders (end-users, procurement agents, etc.). Both textual and graphical documents are the typical means to achieve such communication.

2.3. *Our Approach*

We see the core challenge as being the analysts' need to deal with the *large volume and wide diversity of knowledge* associated with the requirements engineering process.

This knowledge includes domain models, initial requirement conceptions, abstracted views of requirements, formal descriptions of systems, and stereotypical ways to modify these descriptions. ARIES embodies our ideas on how to support analysts to deal effectively with such knowledge. Its design features:

- a modularized central repository of requirements information,
- a single, highly expressive internal knowledge representation scheme,
- communication between analyst and system in terms and styles familiar to the analyst, and
- tool support for the analyst to manage, analyze, and evolve requirements information.

In the remainder of this section we expand upon the motivation for the above design, and briefly touch upon how ARIES fulfills its supportive role; subsequent sections present in more depth the system's capabilities and how they are achieved.

2.4. *Modularized Central Repository*

A central repository encourages the gathering together of all pertinent requirements information into a common, shared framework. Modularization is necessary to structure the large volume of information, for purposes of supporting multiple projects and analysts simultaneously, both to permit sharing of common information, and to permit tolerance of divergent, incompatible information. ARIES offers modularization though structures called *folders*. By developing multiple folders, the analyst(s) can keep separate requirements information. Folders can inherit and import information from other folders, thus providing for the sharing and transfer of information. Analysts control this sharing, and in the course of typical specification development, gradually increase the extent of sharing as inconsistencies are resolved.

2.5. *Highly Expressive Internal Knowledge Representation Scheme*

A single internal knowledge representation scheme provides the agreed-upon representation with which all tools communicate inside the system. We place a heavy emphasis on codification and use of domain knowledge in requirements analysis. Although a number of researchers have identified domain modeling as a key concern (e.g., Greenspan [8]), it is given short shrift in typical practice. Requirements analysis is usually narrowly focused on describing the requirements for a single system. This is problematic if an organization is interested in introducing more than one computer system into an environment, or when the degree of computerization of an organization is expected to increase over time. We have been modeling particular domains within ARIES, and experimenting with reusing such knowledge in the engineering of requirements for multiple systems. ARIES even maintains information about itself—the ARIES metamodel—within this representation. High expressivity is necessary to capture the multitude of forms of requirements knowledge, however it has the drawback of making it harder (or even impossible) to construct completely automatic tools that reason about and otherwise manipulate the stored information.

2.6. *Communication*

Analysts must be able to input and observe requirements in a manner with which they are familiar. If this were not the case, there would be a detrimental disconnection between the concerns residing inside analysts' heads, and the way in which they interacted with the system. To communicate with analysts, the system translates both ways between analyst styles, and the system's internal knowledge representation. Analysts and ARIES communicate through graphical, textual and other structured presentations common to the analysis community (state transition diagrams, taxonomies, decomposition hierarchies, information flow diagrams, as well as formal specification languages). A structured text facility is of use in constructing of on-line informal engineering notebooks, and in maintaining traceability between pre-existing textual requirements documents and the specifications that are the product of an ARIES mediated development.

2.7. *Tool Support*

The system should help analysts in the performance of their tasks, the major ones being management of information (reusing, specializing, adapting, and sharing between analysts and/or across problems), the analysis of the gathered information (through simulation and the like), and the evolution of requirements. This compromise is necessary in part because of our choice of a highly expressive representation, for which completely automatic tools are not available. ARIES provides *evolution transformations*—executable, declarative representations for stereotypical changes to specifications. A library of such transformations is available, and the system assists the analyst in finding the transformation(s) appropriate to achieving the specific changes desired. To support analysis, ARIES offers capabilities to rapidly construct, populate, and exercise executable specifications, deduction mechanisms to propagate information through the specification, and abstraction mechanisms to extract simplified views of system descriptions. In general, ARIES's tools leverage analysts' insight and understanding by automating the repetitive and mechanical aspects, but continuing to rely upon the analysts for direction and control.

III. An Illustration of Use

To convey a sense of the utility of the approach, and to act as an example to anchor the details which will be presented in the later sections, we show a scenario illustrating the use of ARIES. The scenario illustrates what we believe is a typical cycle in the evolution of requirements. While we have *not* tested the use of ARIES to support real-life analysts in their actual activities, we have striven to achieve a measure of reality by picking a real-life complex domain as our major example. We first outline the domain, and then turn to the details of the scenario.

3.1. *The Domain*

Our domain is the Federal Aviation Administration's Advanced Automation System [22]. The goal of the Advanced Automation Program is to develop the next generation of air traffic control systems. Its requirements are imposed by the real world (i.e., are not under our control), and hence we have had to deal with the variety and volume of those requirements. We have captured sections of the AAS requirements specification in ARIES; we have also included information drawn from manuals on flight procedures (e.g., [2]), and from interviews with information processing specialists with the current air traffic control automation system.

3.2. *The Scenario*

The abstract nature of our scenario is as follows.

1. The analyst reviews a partial requirements description, developed via previous interaction with ARIES. The analyst identifies a deficiency in the current requirements.
2. The analyst decides how the requirements must be modified in order to correct the deficiency, and employs an evolution transformation to perform the modification.
3. The analyst reviews the modified requirements that resulted from the previous step, setting the stage for the process to repeat.

The complete Advanced Automation System includes several computer systems, each encompassing a number of functional areas, such as radar data processing, flight plan processing, and traffic flow management. This example focuses on one functional area, the process of transferring control of aircraft between controllers and facilities, known as "handoff." Requirements pertaining to handoff are recorded in several folders, the most important being a folder we appropriately named `handoff`. At the beginning of the scenario, the requirements for handoff have been partially identified and formalized; the analyst needs to review the requirements in this and other folders, in order to identify omissions and inaccuracies.

3.2.1. Reviewing Requirements: ARIES provides a variety of presentations for the analyst to use to view the contents of the `handoff` folder. These presentations are displayed via a graphical interface implemented on the X window system. The figures show the contents of windows generated by the interface—for windows with graphical contents, we show the actual windows; for textual contents, just the text.

Fig. 1 shows part of the contents of the *overview presentation* of the `handoff` folder, summarizing the folder's contents (namely, declarations of *events*, i.e., actions which may be performed by the AAS system or by its environment, and *relations* between objects). Other relevant information, such as the definitions of objects that can participate in the relations and events (e.g., aircraft), happen not to be defined in this folder, but are instead defined in other folders that this folder makes reference to.

Graphical presentations can be appropriate for showing the relationships between definitions. Fig. 2 shows the *event taxonomy presentation* for the event `init-handoff`. Event taxonomy presentations show the relationships between generic

```
                   DURATION: RELATION-DECLARATION
        HANDOFF-IN-PROGRESS: RELATION-DECLARATION
HANDOFF-IN-PROGRESS-ACCEPTED: RELATION-DECLARATION
                    HANDOFF: EVENT-DECLARATION
               INIT-HANDOFF: EVENT-DECLARATION
     AUTOMATIC-INIT-HANDOFF: EVENT-DECLARATION
        MANUAL-INIT-HANDOFF: EVENT-DECLARATION
     TOP-OF-BLOCK-ALTITUDE: RELATION-DECLARATION
          TIME-FOR-HANDOFF: RELATION-DECLARATION
                        ...
          ALERT-CONTROLLER: EVENT-DECLARATION
             ACCEPT-HANDOFF: EVENT-DECLARATION
```

Fig. 1. A portion of the *overview presentation* of the handoff folder.

and specialized event descriptions, in this case, between the following events:

`init-handoff`, which initiates the process of handing off control,

`automatic-init-handoff`, which is performed by the air traffic control system to initiate handoff automatically, e.g., when a aircraft approaches an airspace boundary, and

`manual-init-handoff`, which describes handoffs initiated by controller command.

Each bubble in the diagram has a two-line label; the top line shows the name of the definition, the bottom line the name of the folder in which that definition resides (in this case, they're all in the `handoff` folder). In this static snapshot, names are truncated; the actual interface displays the full names as the mouse is moved over the bubbles. A line between bubbles portraying events indicates the lower event is a specialization of the upper event.

In order to see what requirements are associated with `init-handoff`, its definition must be viewed in detail. For this purpose, natural language is an appropriate medium to use, since it makes it possible to compare formal descriptions directly against informal requirements drawn from natural language documents or acquired from clients. Fig. 3 shows the contents of the *English paraphrase presentation* of `init-handoff`.

3.2.2. Performing a Modification: The description of `init-handoff` in Fig. 3 omits an important detail: it does not allow for the possibility that handoff has been "disabled" (in practice, controllers can explicitly "disable" handoff—both manual and automatic—preventing it from occurring until it has been re-"enabled"). In our scenario, we suppose that the analyst, after viewing the `init-handoff` requirements, realizes this, and now wishes to introduce and make use of a feature to enable or disable handoff of individual aircraft.

This change is an instance of a stereotypical modification step: whenever an event operates on a class of objects, it might be useful to introduce a condition and add it as an enabling condition on initiation of the event. Evolution transformations were developed in order to carry out such stereotypical modifications. The one that applies this case, `define-and-check-enabling-state`, does all of the following.

- It defines two states, an enabled state and a disabled state, for some class of object.
- It defines two new events, one which moves an object from the enabled state into the disabled state, and one which moves an object from the disabled state into the enabled state.
- It adds a new precondition to an event selected by the user, ensuring that the event will not be initiated if it operates on an object in the disabled state.

The analyst may retrieve the appropriate evolution transformation by selecting one of the generic gestures listed on the left side of the window in Fig. 2. These gestures are applicable to any presentation that is rendered graphically; the specific meaning depends upon the types of objects and relationships shown in the presentation. In this case, selection of the gesture "modify" causes ARIES to retrieve and present in a menu all the transformations which have the effect of modifying event declarations (because event declarations are the content of this particular presentation). The transformations themselves are represented within the ARIES knowledge base, and hence the analyst can use ARIES to study the details of particular transformations in order to determine the appropriate one to apply. Once the analyst directs ARIES to apply a particular transformation, further interaction occurs between analyst and system to provide (or override default values for) the transformation's input parameters.

Since ARIES's evolution mechanisms *per se* are not the main focus of this paper, we omit further details of this part of the scenario (but see Section VI for some discussion of how evolution relates to representation, and [25] for our perspective on the role of and support for evolution within the requirements/specification process).

3.2.3. Reviewing the Results of the Transformation: The results of the transformation are changes to the knowledge base of requirements. The analyst can review these results by re-presenting the various views of `init-handoff`, and comparing these to the old versions. For example, generation of a new English paraphrase of `init-handoff` results in the version shown in Fig. 4; note that the precondition of the event has changed. If `automatic-init-handoff` or `manual-init-handoff` were to be paraphrased, a similar change to their preconditions would be seen because they are specializations of `init-handoff`.

The analyst might also wish to view the new states and transitions that have been created. Fig. 5 shows the *state-transition presentation* displaying the states and transitions for the changed version of `init-handoff`.

3.3. Summary of the Scenario

The above scenario shows what we believe is a typical cycle of activity in requirements analysis—the analyst reviews the existing state of the requirements, identifies a deficiency, makes a correcting change, and reviews the changed requirements. The key to this is ARIES's ability to generate presentations in a variety of styles with which the analyst is likely to be familiar. When the analyst makes changes to the requirements (through the application of evolution transformations), the internal representation is modified, and the analyst can then apply any of the presentations to study the changed requirements. In the sections that follow we will show how ARIES's knowledge representation and associated presentation capabilities make this interaction possible.

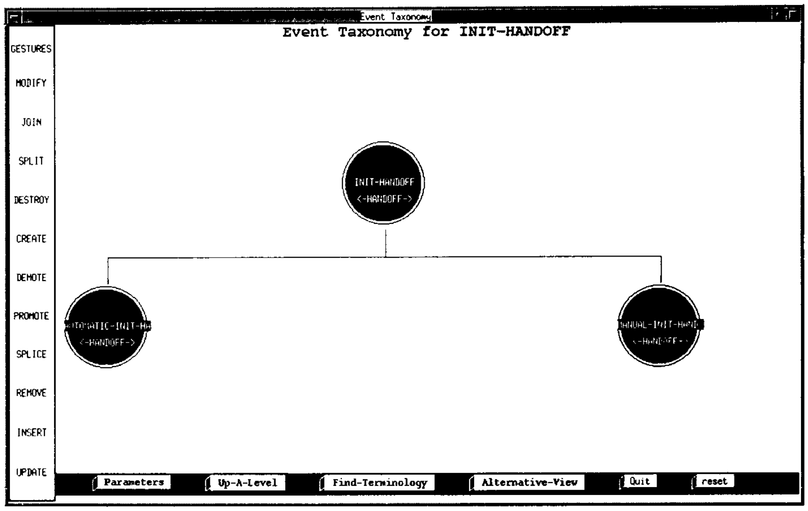


Fig. 2. *Event taxonomy presentation* for `init-handoff`.

```
INIT-HANDOFF is an action of the system. Its sole participant is
a track. To perform an init-handoff, the system sequentially does
the following two steps.
  1. The system asserts that the HANDOFF-IN-PROGRESS relation
     associates TRACK, current-controller and
     receiving-controller.
  2. The system assigns the track-status of TRACK to
     crosstell.

There is a precondition that current-controller must control
TRACK. There is a postcondition that TRACK must be track-status
crosstell.
```

Fig. 3. *English paraphrase presentation* of the `init-handoff` event.

```
INIT-HANDOFF is an action of the system. Its sole participant is
a track. To perform an init-handoff, the system sequentially does
the following two steps.
  1. The system asserts that the HANDOFF-IN-PROGRESS relation
     associates TRACK, current-controller and
     receiving-controller.
  2. The system assigns the track-status of TRACK to
     crosstell.

There is a precondition that current-controller must control
TRACK and disabled must not be true of TRACK. There is a
postcondition that TRACK must be track-status crosstell.
```

Fig. 4. New *English paraphrase presentation* of `init-handoff`.

IV. The Underlying Representation

This section is concerned with how knowledge pertaining to requirements engineering is represented in ARIES. We identify the following as the most important challenges that representation and reasoning capabilities must meet.

- An *extreme breadth of knowledge* must be expressible. Specifications of system behavior, definitions of domain terminology, system organization, nonfunctional characteristics, must all be representable. The system must even represent significant information about itself (in the case of ARIES, a metamodel of the different kinds of requirements objects and their relationships, and also a representation of the evolution transformations used to make changes to the system's information).
- Both *strong and weak expressivity* are needed. Strongly expressive constructs are needed in order to define concepts precisely, especially those that will appear in detailed specifications of behavior and system invariants (e.g., temporal and higher order logical operators). At the same time, analysts use less expressive, but more convenient, notations during initial acquisition of requirements (e.g., the event-taxonomy and state-transition diagrams that we saw in the previous section's scenario).
- *Partial sharing* between representations must be supported to permit multiple projects and analysts to operate simultaneously. For example, it should be possible for one person to work on aircraft handoff, another to work on flight plan processing, and another to work on an entirely different requirements specification—all of which rely on a common body of knowledge.
- *Automated reasoning capabilities* are required. It is generally recognized that highly expressive languages are harder to reason about, both for machines and for people; this paper will show how this problem is mitigated in our approach.
- *A variety of presentations* must be available to view the contents of the internal representation; in fact, it should be possible to define new presentations more or less at will. Thus the representation must not be too closely tied to any one acquisition medium.

4.1. Basic Features of the ARIES *Knowledge Representation*

The basic units of system descriptions in ARIES are *types*, *instances*, *relations*, *events*, and *invariants*. The types, instances, and relations are needed to represent the entity-relationship models common in requirements engineering (e.g., [8], [19]). We designed ARIES's representation to be free from many of the limitations common to representations derived from programming languages or relational data models. Generally

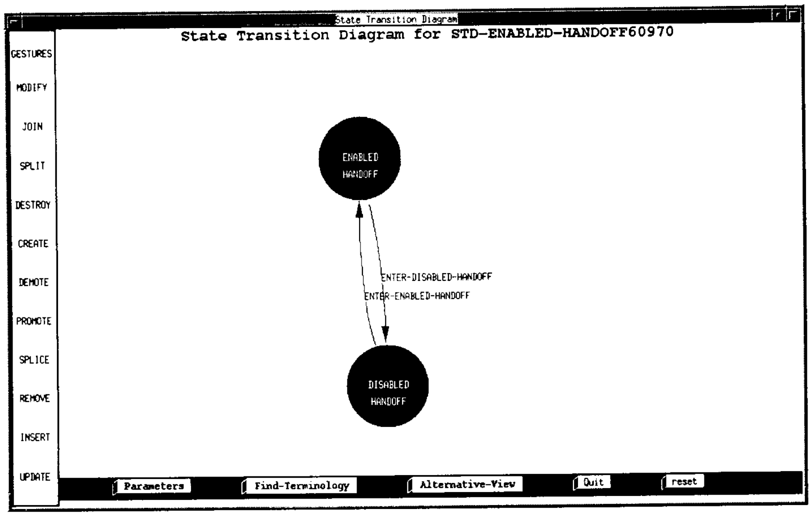


Fig. 5. *State-transition presentation* of `handoff` enablement.

these freedoms make requirements and domain models easier to represent, but make reasoning less tractable.

- Each type can have multiple subtypes and supertypes. In contrast, assuming at most a single supertype makes certain kinds of processing, such as inheritance, easier to compute, it limits the ability to express entity-relationship models of domains.
- Each instance can be an instance of any number of types simultaneously. Again, this freedom is needed in order to represent complex entity-relationship models.
- Relations hold among any types of objects. This contrasts with relational data models, in which relations are defined over "atomic domains"—integers, strings, etc. It also contrasts with frame-based systems in which relations are stored only as slots on some special class of object called a "frame" or a "knowledge base object."
- Relations need not be binary, but can have arbitrary arity. This makes it unnecessary to encode ternary relations as binary relations on some artificial object.
- Relations are fully associative. This contrasts with approaches in which relations must be defined in pairs, one mapping from the domain to the range and one mapping from the range to the domain.
- Types and relations may be primitive or derived. If a type is primitive, it holds for an instance only if the instance is explicitly asserted to belong to the type or one of its subtypes. If a type or relation is derived, it holds whenever a defining predicate associated with that type or relation is satisfied (e.g., the relation flying-at-high-altitude may be derived by defining that it holds on each aircraft whose altitude is above 20 000 ft).
- Events subsume all actions of the modeled system's processes (e.g., the AAS system's `handoff` action), its environment's processes (e.g., movement of aircraft), and ARIES's own processes (e.g., evolution transformations). Events have duration, possibly spanning multiple states in succession, and involving multiple entities of the system. This contrasts with frameworks in which events are assumed always to occur instantaneously.
- Events can have preconditions, postconditions, and methods consisting of procedural steps. They may be explicitly activated by other events, or may occur spontaneously when their preconditions are met. They may have inputs and outputs. However, event definitions can affect the state of the system in ways other than generating outputs: they can assert and remove relations between objects, and create and destroy objects. A variety of behavior models can be mapped onto this event model: for example, state transitions in state transition diagrams are modeled internally as events in ARIES.
- Invariants are predicates that are required to hold, either at all times or whenever a particular event is active. Many requirements on system behavior are naturally expressed as invariants (e.g., aircraft in flight must maintain a minimum separation). Nonfunctional characteristics are also often expressible as invariants (e.g., the required response time of the AAS system must be less than some maximum).

4.2. Example

ARIES's descriptions take the form of a collection of objects, each of which represents some element of the system. These objects participate in relations, relating the objects to other objects or to values such as symbols, strings, or other objects. These relations are stored in a database using the AP5 virtual-memory database system [11].

Fig. 6 shows the definition of `init-handoff`, as displayed by the *describe-object presentation*. This displays the object and its associated objects and values in an attribute-value form suited to human viewing (for a glimpse of the actual machine representation; see Section 5.2.1). The definition is represented as an object of type `event-declaration`. It

```
INIT-HANDOFF: Event-Declaration
Concept description:
   The general definition of the process of initiating handoff
Folder: HANDOFF
Role 0: CURRENT-CONTROLLER
Role 1: RECEIVING-CONTROLLER
Input parameters: TRACK: TRACK
Precondition: UNNAMED
Postcondition: UNNAMED
BODY: UNNAMED
```

Fig. 6. *Describe-object presentation* of `init-handoff`.

has a text string identifying what this definition is, associated via the `concept-description` relation. The object is related to a folder named `handoff`, via a relation called `component`. It is related to two objects of type `role`, named `current-controller` and `receiving-controller`; these are associated with the event declaration via a `role-of` relation. These "role" objects identify objects in the domain, e.g., controllers, that participate in the event. The definition has an input parameter named `track`, associated via the `parameter-of` relation. It has a precondition, associated via the `precondition` relation. The precondition prints as `unnamed`, indicating that it has no `name` relation associated with it.

Fig. 7 show a similar presentation of the precondition of `init-handoff`. It shows that the object is an instance of two types: `enabling-pred`, i.e., a predicate that is a precondition of some event declaration, and `predicate-query`, i.e., it is a predicate which queries whether some relation holds. The relation being queried is `control`, associated via the `concept` relation. The parameters used in the query are `current-controller` and `track`, associated by the `actual-of` relation. This precondition is thus a query testing whether a `control` relationship holds between the current controller and the track.

These examples illustrates key features of this method of representation. First, the representation incorporates both informal information (e.g., concept description text strings) and formal representations (e.g., the precondition query). This contrasts with most conventional requirements environments, e.g., SREM [1], in which the use of formal representation is much more limited, and individual requirements are recorded only as informal text. Second, subcomponents of objects are themselves objects. This fine-grained object representation contrasts with software engineering environments such as CLF [32] in which individual function definitions are recorded as text strings, and parsers or compilers must be invoked in order to interpret them.

4.2.1. Metamodel Relations: Each object in a system description may participate in a variety of relations with other objects. Examples of such relations are `concept-description`, `component`, `role-of`, `parameter-of`, `precondition`, `concept`, `actual-of`, and `generalization`. The complete set of relations supported is quite large, for two reasons. First, the variety of information that must be captured in the ARIES knowledge base, including formalized requirements, informal descriptions, and nonfunctional characteristics, is quite high. Second, relations were included to support the various graphical presentations

```
UNNAMED: Enabling-Pred and Predicate-Query
CONCEPT: CONTROL
ACTUAL:
  CURRENT-CONTROLLER
  TRACK
```

Fig. 7. *Describe-object presentation* of the precondition of `init-handoff`.

in ARIES: for every graphical presentation that depicts links between objects, there is a relation in the metamodel that corresponds to the link. In general, many of the abstract relationships between components of an ARIES description that are useful to support reasoning and processing are represented as relations in the Metamodel.

4.3. Advanced Features of the ARIES Knowledge Representation

4.3.1. Specialization: ARIES makes extensive use of specialization hierarchies, more than is typical of many representation schemes. For example, it supports specialization hierarchies not only of relations, but also events, including the events within the system's own metamodel (e.g., those representing evolution transformations). The key to this is a uniform semantic framework within which specialization is defined consistently for types, relations, and events. This framework is described in detail elsewhere [27]. Briefly, when applied to types, this notion of specialization reduces to simple logical subsumption: if S is a specialization of T, then if x is an instance of S it is also an instance of T. When applied to events, specialization means that the functional requirements on the events are subsumed: if event declaration E is a specialization of event declaration F, then every occurrence e of E also meets preconditions and postconditions of F—for example, `manual-init-handoff` is a specialization of `init-handoff`, and hence inherits the preconditions of the latter. This notion of event subsumption is compatible with other common approaches to event hierarchies such as that of Kautz [28].

4.3.2. Parameterized Concepts: The approach to specialization described above relies upon all concepts having multiple roles, each of which have values assigned to them. Roles may be defined in terms of other roles, e.g., the actor role of `takeoff` is defined to be the same as the input of `takeoff`. It is also possible to define concepts in which roles are declared but unbound. Such concepts are called *parameterized concepts*.

For example, the ARIES knowledge base contains a reusable definition of the concept of "tracker," i.e., a system that tracks the movements of objects such as aircraft. The ARIES representation of tracker has unbound roles representing the accuracy value (the maximum distance between the true location of an object, and the predicted location claimed by the tracker) and type of object being tracked. Any instance of this concept must have those roles bound, e.g., to `100 feet` and `commercial-aircraft`, respectively.

Higher Order and Temporal Operators: Higher order operators are included in order to define properties that hold for entire classes of concepts. Temporal operators permit expression of invariants that hold between system states at different points in time.

An example of a concept that makes use of both is the definition of `first-come-first-served`. This property is higher order because it can hold for any process that handles and acts on requests, i.e., it is a property of a particular class of events. It is defined as a temporal relationship between requests and responses: if two requests of the same server are made in a particular order, the responses must be in the same order.

4.4. Structuring the ARIES Knowledge Representation

The contents of the ARIES knowledge base is organized in *workspaces* and *folders*. Whenever an analyst is working on a problem, it is in the context of a particular workspace. Each workspace consists of a set of folders, and each folder consists of a set of declarations, i.e., type, relation, event, instance, and invariant declarations.

The ARIES folder system was designed to meet two objectives, first, to support partial sharing among analysts working on the same project, or on different projects, and second, to support the reuse of knowledge across multiple projects, especially domain knowledge that may not be specific to any one software project.

Partial sharing is supported by means of a use hierarchy. Each folder uses some set of other folders. The used folders are searched whenever an analyst refers to a concept by name in some presentation, in order to determine the referent of the name. If two concepts with the same name exist in different folders, the analyst may specify which definition takes priority should both folders be used by a third folder.

Approximately 50 folders in the ARIES knowledge base are marked as *reusable*—i.e., they contain information relevant to a variety of system descriptions, not just to one. Domain knowledge, such as properties of aircraft and airspaces, fall into this category, as do general descriptions of activities such as tracking and handoff. One might hope to build a sizable knowledge base containing such information, and then use it in a variety of applications. Unfortunately, it is impossible in general to represent domain knowledge so that it can be usable in an arbitrary application, since the choice of representation depends upon the kind of reasoning about the domain that is embodied in the design of the application. For example, there is no one best way to model the behavior of aircraft in flight. For a radar tracking system, the important aspect of aircraft flight is its continuous dynamic behavior, how aircraft maneuver in real time. For a traffic management system, the individual maneuvers of the aircraft are unimportant; rather, the concern is the overall flight plan, which can be represented as a sequence of straight-line route segments.

In order to address this problem, reusable folders are organized into hierarchies, according to their degree of specificity to a particular task. For example, multiple models of object motion are included in the current ARIES knowledge base, each in different folders. The most general version, in the folder `generic-actions`, is not specific to any domain or problem; some, such as `traffic-light-vanilla-behavior`, are specific to road traffic control, and some, such as `maneuver`, are specific to aviation. Furthermore, different models are defined within the aviation domain: `maneuver` models aircraft as moving in continuous trajectories over time, and `basic-descriptive-aircraft-move` models motion as consisting of straight-line motion from source to destination.

4.5. Self-Representation in ARIES

The general framework for representing knowledge is used in ARIES to represent all domain knowledge in the system, as well as substantial parts of the ARIES system itself. Thus the knowledge base contains definitions of not only domain types (e.g., `aircraft`), but also of types of knowledge base objects (e.g., `type-declaration`). Likewise, in addition to relations between domain types (e.g., the `altitude` relation between aircraft and numbers), the ARIES knowledge base includes relations between knowledge base objects (e.g., the `generalization` relation between concept declarations and their supertypes or generalizations).

This self-representational approach is particularly important for the library of transformations, which are represented as events. For each transformation, the inputs, outputs, preconditions, and postconditions may be specified, just as events in application domains are described. Transformations are organized in a specialization hierarchy according to the same subsumption principle that holds for events in general.

This self-representing approach affords several advantages. Having declarative specifications of ARIES's components makes it easier for engineers to understand what those components are and what properties they have. Tools for describing and explaining knowledge base components, such as the ARIES Paraphraser for generating natural language, may be applied to components of the ARIES system itself.

At the same time, this self-specification approach results in a certain amount of duplication. The ARIES representation is for specifications, not for implementations. Thus for each specified construct there is a corresponding implementation of that construct. For example, for each transformation in the system there is both a declarative specification, represented using the ARIES representation, and an implemented Lisp function that is called in order to execute the transformation. The correspondence between specification and implementation is automatically maintained by always defining transformations by means of special Lisp macros that expand into both partial specifications and Lisp implementations of the transformations.

An example of the use of self-representation is the system's display of the taxonomy of transformations below the transformation `add-relation-with-parameters`, shown in Fig. 8. Transformations lower in the taxonomy are specializations of those higher up, i.e., they achieve a superset of changes. ARIES's generation of this diagram is possible because transformations, elements of the ARIES system, are represented as events with the system's knowledge base, and as such can be manipulated as any other kind of events, in particular, displayed as shown here in an event taxonomy diagram (just as the `init-handoff` event was displayed in Fig. 2).

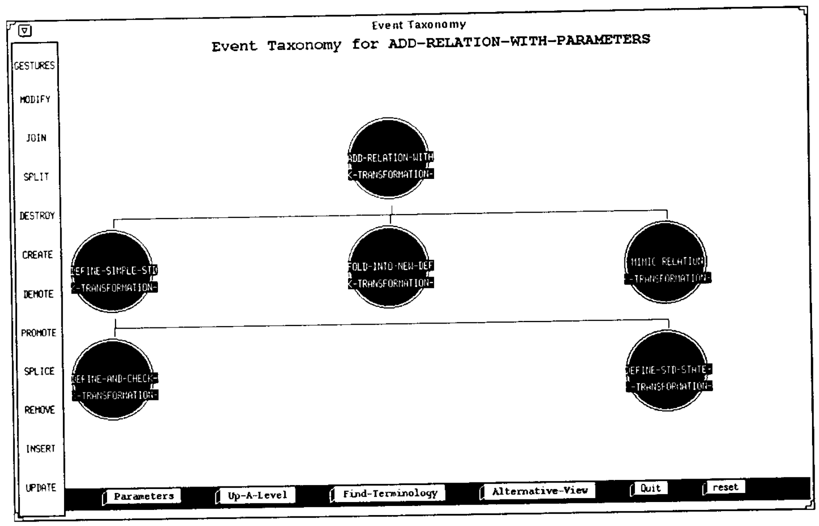


Fig. 8. A taxonomy of transformations.

4.6. Abstraction and Extraction

The sum total of the capabilities outlined above comprise ARIES's knowledge representation scheme. While their design facilitates broad expressivity and thus encourages the capture of a wide range of requirements information, it inhibits automated reasoning and presentation. This is because typical reasoning or presentation tools are able to deal with only a subset of the expressible representations. Furthermore, of the large amount of knowledge represented, for a given purpose only some of it will be relevant. Our response to these issues has been to define specialized representations on top of the general internal representation. These specialized representations abstract and extract the pertinent and manageable information for the purpose at hand. For example, the *event taxonomy presentation* of `init-handoff` (Fig. 2) dealt only with the concepts of *types*, *folders* and their *names* and specialization relationships; furthermore, only the specific instances of those concepts related to `init-handoff` were shown. Thus in generating this presentation ARIES has extracted just the appropriate information to portray. In this subsection we explore how such extraction and abstraction is achieved; consideration of the closely related issue of how to *portray* the extracted information to the analyst is deferred until Section V.

The simplest forms of abstraction and extraction can be defined as retrievals computed from the internal knowledge representation scheme. For example, the transitive closure of relations already existing in the representation, such as `generalization`, can be computed automatically at minimal cost. Similarly, retrieving the names and folders of various concepts is simply a matter of following the internal `name` and `component` relation that links each concept with its name and folder respectively. Three pragmatic classes of problems impede this approach in general:

- retrieval computations may be intractable given our current capabilities,
- the internal representation may contain concepts that are incompatible with the specialized representation, or
- the internal representation may have insufficient information to complete the mapping to the specialized representation in an unambiguous manner.

Intractable Computations: The `state-transition` presentation (as was shown in Fig. 5) is one for which completely automatic computation is intractable within ARIES. The reason for this is that to derive the `state-transition` relations automatically, it would be necessary to derive weakest preconditions and strongest postconditions for arbitrary events, and determine what state relations are implied by those preconditions and postconditions. This computation is extremely hard in the general case, involving symbolic evaluation of the procedural definition of the event. Our previous work on theorem-proving techniques for symbolic evaluation [10] are potentially applicable, but we have found such tools to be difficult to scale up and extend to accommodate temporal and/or higher order operators, and thus within the system they are still limited in their applicability. Likewise, deriving specialization hierarchies of relations, types, and events is difficult in the general case.

We circumvent this difficulty by relying upon the analyst to aid in the construction and maintenance of this difficult-to-compute information. For example, in the case of ARIES's internal representation of its own transformations, organization of these into a specialization hierarchy relies upon assistance from the system creator: ARIES does a partial, conservative analysis (conservative, in the sense that the effects that it deduces are always correct, but are not necessarily complete) and it is our responsibility to explicitly assert those effects that the system has failed to determine through analysis. We find this cooperative processing, in which human and machine both contribute to the reasoning task, to be an effective approach to overcoming the problems of intractable computations.

For the purpose of state-transition presentations, ARIES relies upon the analyst to indicate what is a state and what is not—for example, the analysts may indicate that the second parameter of the `track-status` relation, that links a `track` to its `status` (one of the finite set of values `operational, training, maintenance,` etc.), is the determiner of the state of the `track`; alternatively, the analyst might direct that whether or not a track is in the `dropped` relation is the indicator of a track's state (dropped, or not dropped). Having been given this indication, ARIES maintains the information on states across the application of evolution transformations. For example, the `define-and-check-enabling-state` transformation of 3.2.2 not only changes the internal representation of the events to which it is applied, but also makes the corresponding changes to the state information associated with those events.

4.6.2. Incompatibility: Incompatibility arises when the task is to extract constructs whose realization within the internal representation makes use of constructs not present in the specialized representation. For example, when generating ERSLA (a specification-oriented extension of the very-high-level programming language Refine) from the ARIES representation, incompatibility arises because ERSLA does not support the use of arbitrary relations, but rather uses sets and maps (e.g., in ERSLA one would use maps whose range is the set of booleans, `false, true`, rather than unary relations). Thus in generation of the ERSLA equivalents of ARIES requirements knowledge, ARIES proceeds by first transforming the ARIES representation of the constructs into a subset of the ARIES internal representation that can be readily translated (e.g., ARIES's unary relations are transformed into ARIES maps onto booleans prior to direct translation into ERSLA).

4.6.3. Ambiguity and Incompleteness: Sometimes, specialized representations assume a more complete form of requirements knowledge than may be present in the ARIES knowledge base. For example, the ARIES representation permits analysts to delay commitment as to what category a knowledge base object belongs to; an analyst can introduce a concept (e.g., `flight`), without committing to whether it is a type, event, or some other construct; this incompleteness is a problem for specialized representations that require constructs to belong to specific categories.

Two methods are employed for handling such underspecified constructs, depending upon the particular presentation being employed. One method is to simply omit from the presentation those objects that are not known definitely to be viewable via a presentation. This method is used in the *type taxonomy presentation*: objects that are not declared to be type declarations are not viewable via this presentation. The other method is to assign the knowledge base objects by default to specific categories for the purpose of creating the presentation. For example, in the *Reusable Gist presentation* underspecified constructs are presented by default as instances. The choice of methods depends upon whether the presentation is intended to present all components of a given folder, or only selected components of a folder.

4.7. Integrating Textual and Relational Representations

The relational representation used in the ARIES Metamodel provides a uniformity that facilitates the development of intelligent tools. Each tool can operate on a subset of the relations in the knowledge base, or on abstract relations defined in terms of other relations. Each tool can thus focus on an abstract representation of interest to it.

There are some types of processing for which a relational representation is unsuited. Since semantic networks may contain cycles, traversal of the network must be done carefully to avoid infinite loops. Each relation in the metamodel has type and cardinality restrictions, which must be checked as the knowledge base is updated. This checking can greatly slow processing when massive updates to the knowledge base are being made, as the case when folders are being loaded from files on disk. Transformation processes, such as the translation of the ARIES Metamodel into ERSLA, are most easily expressed as mappings from textual patterns in one language onto textual patterns in another language. This is not possible if the internal representation is a collection of relations instead of text. Finally, saving and restoring sections of the knowledge base to disk files becomes complicated, because it requires translating a nonlinear, cyclical representation into a linear textual form, and reconstructing the same cycles in the restoration process.

In response to these concerns, the ARIES Metamodel has been implemented so that it has both a textual and a relational form. We use the relational database capabilities provided by the AP5 system [11] as the uniform means to access all forms of represented knowledge. However, a significant part of the processing done within ARIES involves grammatical objects (parse trees), for which we use the POPART language processing system; a sizable subset of the types and relations in the knowledge base are in fact realized in the form of parse trees. The key to combining these two facilities is to use POPART's parse trees as *implementations* of AP5's relations. This is possible because AP5 allows developers to select arbitrary data structures to implement relations. Examples of such data structures are trees, hash tables, and linked lists. The original purpose of this capability was to permit the separation of data structure selection and algorithm design. In ARIES, this capability is used to allow AP5 to treat parse trees produced by POPART as implementations of AP5's relations. In order to allow AP5 to treat POPART parse trees as implementations of relations, we had to define an operation on parse trees corresponding to each of the primitive operations on relations, *asserting* a relation, *retracting* a relation, *testing* a relation, etc. Once this had been done, relations implemented as parse trees could then be used just like any other AP5 relation, and tools that operate on the representation need not be concerned with which relations were realized in POPART and which were not, unless those tools relied specifically upon POPART capabilities.

V. PRESENTATIONS

In this section we focus on the presentations that the users interact with, and show how they are defined and linked to the internal knowledge base.

Our fundamental goal is to give the analyst expedient access to a wide variety of requirements information. To attain this goal we needed to provide a variety of presentations and translations between various representations of information.

5.1. Constructing Presentations: Information Extraction, Layout, and Portrayal

Constructing a presentation involves three independent activities:

- *extraction* of a (manageable) subset of objects from the knowledge base to be presented,
- *layout* of the overall display of those objects on the screen, and
- *portrayal* of the individual objects within that overall display.

These activities are shown in Fig. 9. We next consider each of these activities in detail, discuss the ARIES's declarative definition of presentations, and conclude with an example.

5.1.1. Extraction of Objects to be Presented: Because of the large size of the knowledge base, it is inappropriate to present all of its objects at once. Extraction is the process by which a subset of those objects are chosen for presentation. The results are accumulated into an abstract graph structure, which serves as intermediary between the knowledge base and the diagrammatic presentations. The simplicity of these abstract graph structures eases the task of defining the diagrammatic presentations, which need never be cognizant of the full complexity of the knowledge base.

Extraction was discussed in Section 4.6. Some presentations are extended to deal with the problems that may arise during this abstraction process. For example, the taxonomic diagrams present hierarchies as trees, yet internally the hierarchies may be acyclic graphs. The approach in this case is to use the presentation only to extract and portray descendents of a given node, and to ignore some specialization relations if necessary to produce a tree. If the analyst wishes to select a parent of the given node for drawing a new tree, the analyst is presented all options via a menu, and one is selected. Thus it becomes apparent that the underlying representation contains more than the presentation is showing. Another case in point was discussed in Section 4.6.3, where it was mentioned that some underspecified declarations cannot be expressed in Reusable Gist. There the presented form was defined in such a way that it would be evident to the analyst that the underlying representation is different from what the presentation would literally suggest. In general, it is the responsibility of the presentation to handle such cases, and somehow signal to the analyst that key information is missing in the presentation.

5.1.2. Layout of the Overall Display: Layout styles encode the syntactic organizational features of user interfaces. These styles are defined independently of the threads that connect the presentation objects to the information in the knowledge base. Each style captures a particular approach to organizing and aligning data for display on the screen. The following comprise a representative sample of the available layout styles.

- `Flow-diagram`—Directed graphs with labeled nodes and edges, e.g., state-transition diagrams (Fig. 5).
- `Presentation-list`—A linear list of objects, e.g., summarizing the contents of a folder by listing the names and type signatures of all concepts in that folder (Fig. 1).

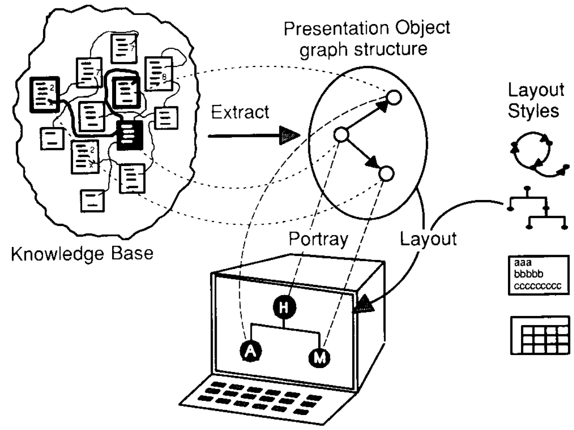


Fig. 9. Construction of `init-handoff`'s event-taxonomy presentation.

- `Matrix`—Tabular presentations of data in row and column format, e.g., spreadsheets of nonfunctional requirements.
- `Pages`—Textual output organized into pages, e.g., informal (natural language) descriptions of concepts in the knowledge base (Figs. 3 and 4).

5.1.3. Portrayal of Individual Objects: The portrayal of individual presentation objects involves icon display and content display. Icons have shape, size, color (or gray scale), and ornamentations. Contents range from very simple name labels to complex textual blocks that are generated through a translation process (the latter will be discussed in Section 5.2).

5.1.4. Declarative Definition of Presentations: In developing ARIES we recognized the need for a wide variety of presentations, and the (continuing) need to be able to easily experiment with, and rapidly modify, these presentations. This led us to choose wherever possible a *declarative* style of interface definition from which the corresponding interface is *generated*. Alternatively, we could have chosen to hard-wire each interface individually. Our selection trades some loss of efficiency and restricted ability to fine-tune presentations in return for rapid construction of modularized interfaces. Presentation objects are organized into hierarchies and we take advantage of inheritance of properties and generic approaches to the creation of interface components. Generation of the code to extract the information is relatively straightforward, because the query itself is readily expressed in the AP5 query language that implements our knowledge base. Generation of the code to portray information on the screen is eased by the fact that it must deal only with the abstract graph structures common to, and tailored for, all the diagrammatic presentations.

5.1.5. An Example of Constructing a Presentation: To illustrate the declarative definition of presentations, and how a presentation of some object(s) is formed, we focus on the *event-taxonomy presentation* and its use in constructing Fig. 2, where event `init-handoff` and its specializations were displayed.

```
(DEFPRESENTATION EVENT-TAXONOMY ()
  :NAME "EVENT TAXONOMY" name of presentation as will appear in menus
  :CONCEPT-DESCRIPTION "SHOWING THE SPECIALIZATIONS OF AN EVENT." menu documentation
  :TOP-LEVEL? T    directs that this appears in top-level menu of presentations
  :GROUPING 'TERMINOLOGY    subgroup of menu in which this appears
  :METHOD 'PROJECTION    currently commentary, i.e., ignored by the system
  :MODE 'GRAPHICAL    currently commentary
  :IMPLEMENTATION-MODEL 'UNBALANCED-TREE    style of this presentation
  :KB-TYPE 'EVENT-DECLARATION    type of knowledge base's "seed" object being presented
  :PO-EDITORS (CONS 'UP-A-LEVEL (TOP-LEVEL-ACTIONS))    edit actions available on
    presentation; as well as all the (standard) top level actions, also includes one to move up a level
  :KB/PO-INTERFACE '(,(MAKE-POID    KB = Knowledge Base, PO = Presentation Object
    :PO-NAME :VERTEX    defines nodes (a.k.a. vertices) of this presentation
    :PO-TYPE '(SHADOWED-CIRCLE)    choice of how objects (nodes) will be drawn
    :KB-TYPE 'EVENT-DECLARATION    type of knowledge base objects being portrayed as nodes
    :NAVIGATION (STANDARD-TREE-NAVIGATORS)    navigations available on each node
    :KB-GETTER 'GENERALIZATION    link to follow in knowledge base to extract objects (nodes)
    :DIRECTION 'REVERSE    direction to follow aforementioned link
    :KB-NAMER #'FOLDER-AND-RELNAMES)))    function that, given a knowledge base
                                          object, generates label for corresponding node
```

Fig. 10. Definition of Event Taxonomy presentation

Fig. 10 shows the definition of the event-taxonomy presentation; SMALL CAPITALS show the code[2], *italics* show comments that we have added for this paper.

When applied to a particular knowledge base object, this declaration provides the information necessary to guide all three activities involved in constructing the presentation of that object.

Extraction: GENERALIZATION is indicated as the relation whose transitive closure is computed to determine the information to be extracted. Because we want the *specializations* of the "seed" event, it is indicated that it be used in the REVERSE direction. Thus when applied to the "seed" event `init-handoff`, this process finds its specializations, their specializations, and so on (in fact, our current knowledge base does not contain any specializations beyond the level of `manual-init-handoff` or `automatic-init-handoff`).

Layout: UNBALANCED-TREE is indicated as the choice of layout style, which causes the display of this abstract graph structure as a (not necessarily balanced) tree. Applied to `init-handoff`, this puts the node representing `init-handoff` at the top, linked by the branches to its immediate specializations `automatic-handoff` and `manual-init handoff`; if there had been further specializations of these concepts in the knowledge base, they also would have been displayed.

Portrayal: SHADOWED-CIRCLE is indicated as the choice of how to portray the event objects. Function FOLDER-AND-RELNAMES is indicated as the function to be used to generate the label for each such object. Thus `init-handoff`, etc., are drawn as shadowed circles, labeled with the name of the event, and the folder in which they reside (namely, the `handoff` folder).

[2]Actual code shown in almost all its ugly detail—for clarity here, we dropped the CommonLisp package prefixes, and changed a few names to begin with KB, standing for the term Knowledge Base as used throughout this paper.

5.2. Translation

Translation is used to input information into the ARIES's knowledge base, and to generate the textual contents of various presentation forms. The external languages that ARIES connects with through translation follow.

English—Concepts of our internal specification language can be described in natural language by our paraphraser tool. This is for output only—the system does *not* do natural language understanding.

Reusable Gist—The behavior-oriented aspects of the specification are represented in Reusable Gist, a version of our in-house specification language Gist that we have used for many years [3]. The system automatically translates its internal representation of specification knowledge into an external Gist form which is much more palatable to human comprehension. This translation goes in both directions—it is also possible to enter or modify requirements and specification information by providing or modifying the external form of Reusable Gist text, which the system parses and translates into its internal representation.

Other formal languages—As discussed earlier, we need to output portions of our specifications in the specification language ERSLA, ready for the overall KBSA effort's follow-on phase of mechanical optimization. Also, we wished to input information from knowledge bases built in the knowledge representation language Loom [29] (a classification-based knowledge representation language commonly used within the AI community). In particular, we imported the Penman Upper Model [4]—thus saving ourselves the effort of recoding it from scratch. To do these imports and exports we built one-way translators, from Loom to ARIES, and from ARIES to ERSLA.

5.2.1. Translation Mechanisms: To provide translations between ARIES's internal representation and the other formal languages, we use Wile's POPART system as the basis for defining translators. POPART, when provided with a BNF-like description of two languages, produces (among other

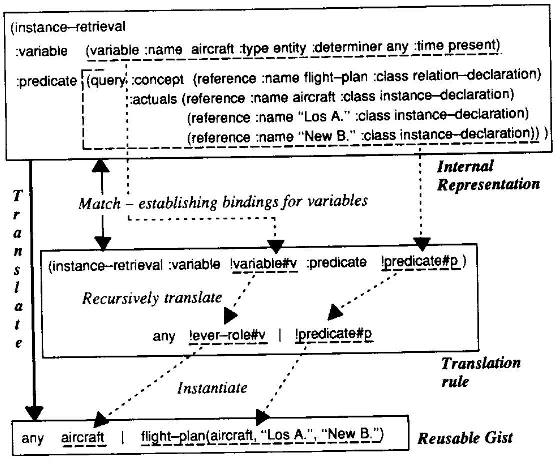


Fig. 11. Translation from the internal representation to Reusable Gist.

things) a capability for easy definition of translators between those languages (see [41] and [42] for details). We use these capabilities extensively.

As illustration, Fig. 11 shows the translation of an `instance retrieval` concept into its equivalent Reusable Gist form. The internal representation is shown in the top box of the figure, and its external Reusable Gist equivalent is shown in the bottom box. As can be seen from this small example, the internal form is quite verbose, demonstrating why this representation is for machine processing, not human perusal! The translation rule for instance retrievals, shown in the middle box, takes the form of an input pattern (its top line) in the internal representation language, and an output pattern (its bottom line) in the Reusable Gist language. Translation proceeds by matching the input against the pattern; bindings established by this are then recursively translated, and the results instantiated in the output pattern to realize the result, the Reusable Gist shown in the bottom box of the figure. This recursive translation process is a feature of POPART's mechanisms, and saves us the tedium of explicitly coding the recursive translation of substructures in many of the translation rules.

Our complete translator from the internal representation to Reusable Gist consists of a set of such rules, one for each type of construct in the grammar of the internal representation. We have similar rule sets to do translation in the reverse direction (i.e., from Reusable Gist to the internal representation), and to generate executable code for simulation as part of the validation component (Section 6.2).

In defining the translation rules for producing Reusable Gist, we invested some extra effort to cause them to use notational abbreviations where possible. By notational abbreviations we mean the syntactic shorthands that permit a more concise expressive form for certain forms of expression, for example, writing addition as in infix expression rather than as a retrieval from a ternary relation (e.g., "`3 + 4`" rather than "`any integer | plus(3,4,integer)`"). Reusable Gist offers a number of such abbreviations, whereas the internal representation eschews these in favor of a more uniform but verbose style. A naive translation of the internal form that did nothing in this direction would, we conjecture, produce unsatisfactory output. We feel that our policy has the further advantage of ensuring that Reusable Gist displayed by the system always makes consistent use of these abbreviations.

We have opted for canonical layouts and translations to provide uniformity and to postpone dealing with some of the engineering issues involved in mixing user-initiated formats with automatically generated formats. If the analyst manually rearranges the layout of a diagram, ARIES maintains data structures which enable it to present the revised layout until modifications to the knowledge base dictate that the presentation be updated. At this point, the user-initiated positions are discarded. A similar capability exists for translations. To preserve the exact form of Reusable Gist as entered by the analyst would require retention of information that we currently discard. Our choice to discard such information is a compromise, insofar as we may imagine a more sophisticated system that retained such stylistic choices, but its realization would further complicate the representations[3].

5.2.2. Generation of Natural Language: For completeness, we briefly discuss ARIES's generation of natural language paraphrases. The tool that does this was originally developed as part of the Knowledge-Based Specification Assistant, one of the precursors to ARIES, and has been described extensively elsewhere [31], [38]. Pertinent to this paper is the point that the paraphraser is treated as just another tool to convert between representations.

We also note that the paraphraser was originally constructed to operate on what evolved into the ARIES metamodel representation (in contrast to, say, hard-wiring it to the surface syntax of the specification language). This proved to be beneficial when the paraphraser was later converted to be part of the Concept Demonstration project, and to operate on the ERSLA language, as reported in [43]. This lends credence to our faith in the utility of the metamodel foundations of the internal representation.

5.3. Summary of Presentation Issues

In summary, to populate ARIES with a wide variety of flexible presentations, we use a declarative style of definition, in which the system builder selects from a number of provided building blocks to compose the features of the presentation he/she desires. We have chosen to separate the phases of i) extracting the subset of the knowledge base information to portray, and ii) displaying that information in an appropriate form. We have found that:

- the underlying knowledge representation is suited to defining extraction of knowledge; previous work involving the natural language paraphraser reported similar advantages;
- we were able to define a variety of presentation styles, making use of the abstracted form of the information as an aid to their construction;

[3] Furthermore, we may imagine an *ideal* system that, in response to changes in the information being portrayed, would incrementally adjust an analyst-tuned presentation to remain in keeping with the analyst's style!

- we were able to define a variety of diagrammatic presentations defined declaratively in terms of those styles; and
- use of POPART's grammar support tools facilitated our construction of several automatic formal-language to formal-language translators.

We have consistently maintained an emphasis on flexibility (ease of definition and subsequent modification of presentations), at the expense of a less efficient, and not as finely tuned, interface. Until we have more confidence in the precise nature of presentations best for supporting the analysis process, we feel constrained to make such choices.

Translation is used to connect the ARIES internal representation and various external representations. We make use of the POPART system to provide the mechanisms from which to build translators. Our ability to use POPART's parse-tree representations is crucial to this, and reinforces the importance of having linked these representations with the relational database, as was discussed earlier in Section 4.7.

VI. Evolution and Analysis

In the first section we identified *evolution* and *analysis* as important activities within the requirements/specification process, alongside acquisition and presentation. In this section we briefly summarize our work toward their support, and how it relates to the issues of representation and reasoning.

6.1. Evolution

ARIES supports evolution through its diagrammatic presentations—the user conducts manipulations that are meaningful and intuitive with respect to the presentation currently being viewed, and the system makes the corresponding changes to the underlying representation. This was briefly alluded to in the state-transition diagram development example of Section III. We now outline the mechanisms that make this possible.

The ARIES system currently has over 180 evolution transformations—operators that modify system descriptions in a controlled fashion (e.g., `define-and-check-enabling-state` that we saw applied in the example scenario to define and use enabling conditions on an event). In making these available to the analyst, we have had to avoid two pitfalls that would have rendered them unusable. First, it would be inappropriate to expect the analyst to select from a linear menu of these transformations, or even from submenus coarsely organized into general categories. Second, it would be inappropriate to hard-wire transformations to editing actions, since the analyst may have only a rough idea of the modification desired and can benefit from ARIES's assistance in accessing a meaningful collection of related transformations.

To avoid these pitfalls, ARIES has been constructed to support the following mode of interaction: The analyst selects the appropriate transformations through indicating the desired *effects* on a given presentation, and the system retrieves the transformation(s) that will or may achieve those effects—typically, this will be a small subset of all possible transformations, and it is much more reasonable to expect the analyst to make further selections as necessary from this subset. Furthermore, the transformations are organized in a specialization hierarchy, following the same principles for event subsumption as that described in Section 4.3.1—this makes sense because transformations are simply a category of events, and are modeled explicitly as such in the ARIES knowledge base. The system is able to perform this retrieval because each evolution transformation is characterized by its *effects* on each of the dimensions of semantic properties. The fundamentals of our approach are as follows.

Effect descriptions—a goal-oriented description which characterizes modifications in terms of their effects on each of several semantic dimensions. The effect description language is less expressive than the full precondition and postcondition representation of ARIES, but expressive enough to support detailed retrievals and classifications.

Linking evolution transformations to effects—each evolution transformation is characterized by the effects that it achieves. ARIES is able to analyze evolution transformations to determine some of their effects. This analysis is, when necessary, supplemented by developer-provided information.

Linking presentations to effects—each presentation has available certain obvious and intuitive manipulations, known as *gestures*. The analyst's chosen gestures are linked to the corresponding effects, and these in turn are used to select transformations.

The benefit of this staged approach is that the analyst can directly modify presentations in an intuitive manner (e.g., create a node in a state-transition diagram), and have the system make the corresponding changes to the underlying knowledge representation (e.g., create a representation of a new state corresponding to the state-transition diagram's new node). Also, this multistage approach makes it far easier to add new presentations and new transformations.

The keys to making this possible are the capabilities provided by the underlying representation language, and the use of this language and the tools that operate on it to represent and analyze the transformations themselves. The reader seeking further details of this aspect of the system is referred to [26].

6.2. Validation

While presentations help us to understand and develop specifications, we need additional tools to inform us about the dynamic behavior of specifications, and help us to validate emerging specifications. Validation at this level is often characterized as validation with respect to the client's or stake-holder's intent. The goal of validation is to identify those aspects of the specification which do not conform to the client's intent and then to make appropriate changes. More prosaically, this boils down to uncovering errors in the specification and fixing them.

In ARIES we use simulation to provide validation of dynamic behavior. However, for most realistically sized application domains, the following factors inhibit tractable simulation used for validation.

- The specification, particularly at the early stages of the requirements acquisition and specification development process, may contain undefined or partially defined terms.

- The specification may not be directly compilable into a sufficiently efficient implementation.
- Once compiled, the simulation of a specification may produce voluminous amounts of data, so voluminous that it is effectively incomprehensible.

In ARIES, we mitigate these problems by decomposing the validation process into a collection of validation activities, each of which revolves around resolving a *Validation Question*. Such a focus allows the analyst to abstract out of the specification those details unrelated to the current validation question. This results in a smaller specification which is much more tractable—it will have far fewer undefined or partially defined terms, it will compile into a more efficient simulation, and the details of the simulation will be only those pertinent to answering the particular validation question. See [5] and [6] for more details. Note the similarity between abstraction and extraction for the purposes of analysis, and for the purposes of presentation as discussed earlier in Section 4.6. Many of the same problems, and responses to those problems, arise in constructing specifications for simulation.

We are also working toward visualization of the running simulation through animated presentations that also have used these techniques to provide focus on specific portions of the overall behavior.

VII. Related Work

Several other efforts have investigated connecting particular instances of alternative presentations of the same information. For example, Fraser *et al.* [15] connect a graphical presentation of requirements (Structured Analysis) to the formal language of the Vienna Development Method; similarly, Dick and Loubersac [12] connect Entity-Structure Diagrams and Operation-State Diagrams to VDM. ARIES goes beyond these by providing support for building such connections (the POPART translation mechanisms—Section 5.2.1, and the linkage of parse-tree representations to the AP5 relational database representation—Section 4.7), and has used this to construct multiple such connections.

Some CASE tools, such as STATEMATE, also support multiple notations. Where ARIES differs from these systems is that in CASE tools the notations are required to convey distinct information, so that edits to one diagram do not result in changes to other diagrams. In ARIES the information conveyed in different presentations may overlap. For example, the Reusable Gist presentation describes many aspects of a system that can also be presented by narrower presentations such as information flow diagrams or state transition diagrams. In this respect ARIES is similar to the PECAN system, which allows programs to see textual and flow-chart views of programs at the same time [37]. Where ARIES differs is that it allows analysts to edit most of these presentations, and edits to one presentation can result in changes to other presentations. In PECAN only the textual presentation can be edited, and other views are read-only. ARIES makes use of a general mechanism for mapping presentations onto a common underlying representation. This scheme makes it possible to provide multiple editable notations, and map changes onto the underlying representation as well as onto other notations.

Perhaps the work closest to ARIES is that of the PRISMA project [34] (also a system for assisting in the construction of specifications from requirements), and the "viewpoints" framework [14] (essentially a generalization of the ideas embodied in PRISMA). Like ARIES, these efforts stress the use of multiple presentations of the (emerging) specification, graphical displays of these, and a semantic-net formalism underlying each (in PRISMA, the presentations—"views" in their terminology—that they have explored are data-flow diagrams, entity relationship models, and petri nets). Unlike ARIES, they do *not* employ a single, all-encompassing internal representation, rather, they advocate the use of pairwise connections between the representations. This necessitates the definition of n^2 connections in order to completely connect n presentations, whereas in ARIES we need define only n (one connection to the internal representation for each of the n presentations). Also, ARIES's tools support the construction of these connections, whereas these other efforts appear to have defined the connections in a more ad-hoc manner. Finally, ARIES has gone further in support of evolution, as discussed in comparison to PECAN. Conversely, PRISMA has gone further toward defining heuristics that operate upon their presentations (e.g., consistency and completeness checkers, both within single presentations, and between pairs of presentations), an aspect that we have only recently begun to address.

There are some other aspects of our approach that we feel deserve brief comparison with other work.

Expressiveness—Swartout and Smoliar [40] have argued for expressiveness in representations—even at the risk of falling over Brachman and Levesque's "computational cliff" [9]—when representing the knowledge underlying software design. ARIES is an illustration of the validity of this position. Work that takes the opposite tack, of limiting expressiveness in order to retain computational tractability, is typified by the CLASSIC system [7], and the COMET system [30]. In both cases languages with limited expressiveness are employed in order to facilitate automatic classification. Generally, the focus of these efforts is on dealing with descriptions that capture some, but by no means all, aspects of the objects in question (software components and the like). In contrast, ARIES aims to capture more complete descriptions (e.g., we must at least capture enough information to be able to generate specifications from which to derive implementations). Our need for high expressivity means we have had to abandon the guarantee of fully automatic analysis of the contents of our knowledge base, and we rely upon human fine-tuning of the approximate analysis that our tools are able to perform. The crux of the distinction is whether one aims to deals with only partial descriptions, or with complete specifications.

Interchange—The goal of having a representation that can be mapped onto a variety of different representations is also the motivation for the Knowledge Interchange Format effort [16]. Our emphasis is on a common internal knowledge representation instead of a representation for transmission of knowledge, but the goal of a common denominator of linguistic expression is the same.

Structuring—In applying ARIES to representing portions of air traffic control requirements, we encountered the need to deal with, particularly in regard to structure, large volumes of knowledge. Our approach to structuring has much in common with others investigating sharable, reusable knowledge bases [33]. Folders in ARIES bear some similarity to microtheories in CYC [18], and to Doug Smith's domain theories. The use of workspaces as a metaphor for development or representations, and the mediation of by intelligent tools via interactive diagrammatic presentations, has been explored by Terveen [40].

Group support—ARIES's structuring mechanisms (folders and workspaces) allow multiple analysts to keep separate their own context of work while giving them continued access to shared knowledge, but overall ARIES provides little in the way of support for group activities such as project management, software process modeling, negotiation, and coordination among multiple analysts, co-authoring, capture of discussions and rationales arising from group meetings, and the like. This is the broad area of focus of the DAIDA framework [20]. More specifically, see [13] for analogy heuristics that identify and reconcile differences between multiple analysts' expressions of (supposedly) the same requirements. Techniques such as these, that rest upon a shared representation, should be quite compatible with the ARIES model.

VIII. Summary

This paper has described the approach to representation and presentation of knowledge used in ARIES in support of domain analysts in modeling the target domains for software development, and systems analysts in entering and formalizing system requirements. Our approach can be viewed as applying the notion of a presentation architecture to the domain of software engineering, and incorporating a strong coupling to a transformation system.

The key feature of the approach is to have a single highly expressive underlying representation, interfaced simultaneously to multiple presentations, each with notations of differing degrees of expressivity. This allows analysts to use multiple languages for describing systems, and have those descriptions yield a single consistent model of the system. We have also attempted to support automated reasoning on this underlying representation, without sacrificing expressiveness.

In the matter of interfacing multiple notations to a single representation, we can point to several accomplishments. Our architecture makes it quite easy to generate different presentations, and to define new kinds of presentations. For a wide range of notations, it is also easy to define multiple editable presentations, so that analysts can enter information in many different ways. The approach works because we have provided support both for defining abstractions and projections of the underlying knowledge base, and for portraying such information in various styles.

There are some limitations to the approach which still must be overcome. One problem with our approach is that it focuses on recording the underlying semantics of presentations, instead of details of their syntactic form. For example, when an analyst constructs a flow diagram, ARIES does not record the exact position of each node in the diagram, so if the diagram is recreated later it may not have quite the same shape. This problem could be readily eliminated by including layout information as part of the representation when the analyst has explicitly chosen a particular layout.

Another problem arises when a diagram, or an edit to a diagram, cannot be mapped onto the underlying representation in a unique way. This problem arises whenever multiple expressions in the underlying representation all are expressed in the same way in a given notation. Three approaches have been employed to cope with this problem: 1) to assume a canonical internal form to map from the external notation to the internal representation, 2) to perform modifications via transformations that operate on the underlying representation in unambiguous ways, and 3) simply to disallow editing of certain presentations. None of these approaches is suitable in all cases. Furthermore, there is reason to believe that a fourth approach might be better, namely to delay the translation process until a commitment to a particular underlying form is necessary.

The technique of supporting multiple semantic abstractions and projections also enables us to have automated reasoning capabilities without sacrificing expressiveness. Nevertheless, the automated reasoning capabilities in ARIES are weaker than they could, or should, be. Some automated reasoning capabilities could be attained simply by making greater use of automated reasoners such as Loom. If there is a limitation, it is that the underlying semantic representation supports intractable inference for which there is no counterpart in the abstracted representations. For example, determining whether one specification is a specialization of another is intractable in general. From our point of view, this is not really a limitation. If a person must occasionally verify that one specification is a specialization of another, that is not a problem, as long as the machine can automatically derive specialization relationships in other cases. What we hope to attain, and what we believe we have attained, is reasoning that employs an appropriate combination of human capabilities and human insight.

Acknowledgment

C. Rich has given helpful advice to this project. The authors would like to acknowledge current and previous members of the ARIES project: K. Benner, J. Myers, K. Narayanaswamy, J. Runkel, and L. Zorman. K. Benner helped significantly in the preparation of this document. Considerable feedback from the editors and their cadre of reviewers has also helped clarify our presentation.

References

[1] M. Alford, "SREM at the age of eight: The distributed computing design system," *IEEE Computer*, vol. 18, Feb. 1985.
[2] ASA, *Airman's Information Manual*, Aviation Supplies and Academics, 1989.

[3] R. Balzer *et al.*, "Operational specification as the basis for specification validation," in *Theory and Practice of Software Technology*, pp. 21–49. Amsterdam: North-Holland, 1983.

[4] J. Bateman, "Upper modeling: Organizing knowledge for natural language processing," in *Proc. 4th Int. Nat. Lang. Generation Workshop*, June 1990.

[5] K. Benner, "Using simulation techniques to analyze specifications," in *Proc. 5th KBSA Conf.*, pp. 305–316, 1990.

[6] ——, "The ARIES Simulation Component," submitted to the 7th Annual RADC Knowledge-Based Software Engineering (KBSE) Conference.

[7] A. Borgida *et al.*, "CLASSIC: A structural data model for objects," in *Proc. ACM SIGMOD '89 Conf.* , June 1989.

[8] A. Borgida, S. Greenspan, and J. Mylopoulos, "Knowledge representation as the basis for requirements specifications," *IEEE Computer*, vol. 18, pp. 82–91, 1985.

[9] R. Brachman and H. Levesque, "The tractability of subsumption in frame-based description languages," in *Proc. Third National Conf. Artificial Intelligence*, pp. 34–37, 1984.

[10] D. Cohen, "Symbolic execution of the Gist specification language," in *Proc. 8th Int. Joint Conf. Artificial Intelligence,*, pp. 17–20, Aug. 1983.

[11] ——, *AP5 Manual*. USC-Information Sciences Institute, June 1989. (draft).

[12] J. Dick and J. Loubersac, "Integrating structured and formal methods: A visual approach to VDM," in *Proc. 3rd European Software Engineering Conf.,*, pp. 37–59, 1991.

[13] J. C. S. do P. Leite and P. A. Freeman, "Requirements validation through viewpoint resolution," *IEEE Trans. Software Eng.,*, vol. 17, pp. 1253–1268, Dec. 1991.

[14] A. Finkelstein, J. Kramer, B. Nuseibeh, and L. Finkelstein, "Viewpoints: A framework for integrating multiple perspectives in system development," *Int. J.Software Engineering and Knowledge Engineering*, 1992 (to appear).

[15] M. D. Fraser, K. Kumar, and V. K. Vaishnavi, "Informal and formal requirements specification languages: Bridging the gap," *IEEE Trans. Software Eng.*, vol. 17, pp. 454–466, May 1991.

[16] M. Genesereth *et al.*, "Knowledge Interchange Format," *Tech. Rep. Logic-90-04*, Stanford University, 1990.

[17] C. Green *et al.*, "Report on a knowledge-based software assistant," in *Readings in Artificial Intelligence and Software Engineering*, C. Richard and R. Waters, Eds. Los Altos, CA: Morgan Kaufmann, 1986.

[18] R. V. Guha and D. B. Lenat, "Cyc: A midterm report," *AI Magazine*, vol. 11, pp. 32–59, 1991.

[19] J. Hagelstein, "Declarative approach to information system requirements. *J. Knowledge-Based Systems*, vol. 1, pp. 211–220, Sept. 1988.

[20] U. Hahn, M. Jarke, and T. Rose, "Teamwork support in a knowledge-based information systems environment," *IEEE Trans. Software Eng.,*, vol. 17, pp. 467–482, May 1991.

[21] D. Harris and A. Czuchry, "KBRA: A New Paradigm for Requirements Engineering," *IEEE Expert*, vol. 3, 1988.

[22] V. Hunt and A. Zellweger, "The FAA's Advanced Automation System: Strategies for future air traffic control systems," *IEEE Computer*, vol. 20, pp. 19–32, Feb. 1987.

[23] W. L. Johnson, "Deriving specifications from requirements," in *Proc.10th Int. Conf. Software Engineering*, pp. 428–437, 1988.

[24] ——, "Specification as formalizing and transforming domain knowledge," in *Proc. AAAI Workshop on Automating Software Design*, pp. 48–55, 1988.

[25] W. L. Johnson and M. S. Feather, "Reusable Gist language description," available from USC/ISI, 1991.

[26] ——, "Using evolution transformations to construct specifications," in *Automating Software Design*, pp. 65–92, AAAI Press, 1991.

[27] W. L. Johnson, M. S. Feather, and D. R. Harris, "Integrating domain knowledge, requirements, and specifications," *J. Syst. Integration*, vol. 1, pp. 283–320, Nov. 1991.

[28] H. A. Kautz and J. F. Allen, "Generalized plan recognition," in *Proc. AAAI-86*, pp. 32–37, 1986.

[29] R. Mac Gregor, *Loom Users Manual*, 1989.

[30] W. Mark, *IEEE Trans. Software Eng.*, this issue.

[31] J. J. Myers and W. L. Johnson, "Toward specification explanation: Issues and lessons," in *Proc. 3rd Annual RADC Knowledge-Based Software Assistant (KBSA) Conference 1988*, pp. 251–269, 1988.

[32] K. Narayanaswamy and N. M. Goldman, "A flexible framework for cooperative software development," *J. Syst. Software*, Oct. 1991.

[33] R. Neches *et al.*, "Enabling technology for knowledge sharing," *AI Magazine*, pp. 36–56, Fall 1991.

[34] C. Niskier, T. Maibaum, and D. Schwabe, "A look through PRISMA: Toward pluralistic knowledge-based environments for software specification acquisition," in *Proc. 5th Int. Workshop on Software Specification and Design*, pp. 128–136, 1989.

[35] "The KBSA Project. Knowledge-Based Specification Assistant," Final report (available from USC/Information Sciences Institute), 1988.

[36] *Refine User's Guide*, Reasoning Systems, Palo Alto, CA. 1986.

[37] S. P. Reiss, "Pecan: Program development systems that support multiple views," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 276–285, Mar. 1985.

[38] W. Swartout, "Gist English generator," in *Proc. Nat. Conf. Artificial Intelligence*, pp. 404–409, 1982.

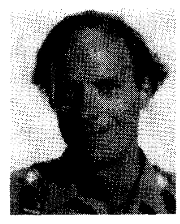
[39] W. Swartout and S. Smoliar, "Report from the frontiers of knowledge representation," *USC/ISI Tech. Rep.*, 1988.

[40] L. Terveen, "Person-computer cooperation through collaborative manipulation," *Tech. Rep. ACT-AI-048-91*, MCC, 1991.

[41] D. S. Wile, "Integrating syntaxes and their associated semantics," available from the author at USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292, USA—wile@isi.edu.

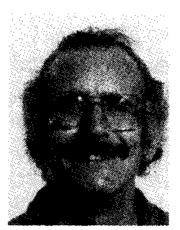
[42] ——, "Organizing programming knowledge into syntax-directed experts," in *Proc. Int. Workshop on Advanced Programming Environments,*, pp. 551–565, 1986.

[43] G. B. Williams and J. J. Myers, "Exploiting metamodel correspondences to provide paraphrasing capabilities for the KBSA Concept Demonstration project," in *Proc. 5th Annual RADC Knowledge-Based Software Assistant (KBSA) Conf.*, pp. 331–345, 1990.



W. Lewis Johnson received the A.B. degree in linguistics from Princeton University in 1978, and the M.Phil. and Ph.D. degrees in computer science from Yale University in 1980 and 1985, respectively.

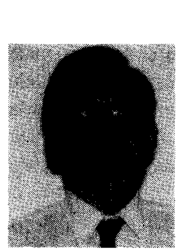
He is Project Leader at the Information Sciences Institute of the University of Southern California and is a research assistant professor in the USC Computer Science Department. He is interested in applying artificial intelligence techniques in the areas of systems engineering and design and in computer-based training. His research interests focus on the development of computer systems that can collaborate with people in order to solve problems.



Martin S. Feather received the B.A. and M.A. degrees from Cambridge University, England, in 1975 and 1976, respectively, and the Ph.D. degree in artificial intelligence from Edinburgh University, Scotland, in 1979.

In October 1979, he joined the Information Sciences Institute, University of Southern California, where he has since become a Research Scientist working on program specification and transformation. His research interests are in formalizing and providing mechanized support for the programming process.

Dr. Feather is a member and secretary of IFIP Working Group 2.1.



David R. Harris received the B.S. degree in mathematics from Cleveland State University in 1966, and the M.S. degree in mathematics from Northeastern University in 1968.

He joined Lockheed Sanders in 1981, and is now a mathematician, developing knowledge-based techniques for improving system and software engineering. He has been a principal investigator on projects ranging from specialized support for automatic test equipment engineering to general purpose support for requirements acquisition and analysis. His research interests include tools for user interface design and construction, and the use of automated deduction to assist in complex analysis and engineering tasks.